UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

(2)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

DTIC
ELECTE
SEP 27 1989
S D
B

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
AD-A212 533	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 89-1253
Kestral Institute — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
6c. ADDRESS (City, State, and ZIP Code) 1801 Page Mill Road Palo Alto, CA 94304	7b. ADDRESS (City, State, and ZIP Code) Building 410 Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR — 8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-88-C-0033
8c. ADDRESS (City, State, and ZIP Code) Building 410 Bolling AFB, DC 20332-6448	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO A2; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) See Cover
~~Theory of Design and Knowledge Based Software Engineering~~

12. PERSONAL AUTHOR(S) Dr Cordell Green

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM 1 Dec 87 TO 31 May 89	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes recent activities on the development of a formal model of the design and evolution of software. The model is intended to be both descriptive and prescriptive. It is descriptive in that we are attempting to capture known design processes in the model. The model is also intended to be prescriptive in the sense that it provides the conceptual basis for the sophisticated knowledge-based software design environments of the future. It should have the *flexibility* to support a variety of design methodologies, be *comprehensive* enough to encompass the gamut of software lifecycle activities, and be *precise* enough to provide the conceptual foundations for an open yet rigorous development environment.

We also present recent work on the structure and design of a class of algorithms called *global search*. The design tactic for global search algorithms provides a rich example of the kind of design process that the abstract model is intended to capture. We present a tactic for designing global search algorithms and illustrate it with the derivation of an algorithm for enumerating cyclic difference sets - a rare kind of set that bear some similarities to the prime numbers. The design tactic has been implemented and used to derivation dozens of global search algorithms including one for enumerating cyclic difference sets.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr Abraham Waksman	22b. TELEPHONE (Include Area Code) (202) 767-5027 — 22c. OFFICE SYMBOL NM

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

89 9 27 023

Toward a Formal Model of the Design and Evolution of Software

AFOSR-TR- 89-1253

Douglas R. Smith
Kestrel Institute
1801 Page Mill Road
Palo Alto, California 94304-1216
December 20, 1988

ABSTRACT

This report summarizes recent activities on the development of a formal model of the design and evolution of software. The model is intended to be both descriptive and prescriptive. It is descriptive in that we are attempting to capture known design processes in the model. The model is also intended to be prescriptive in the sense that it provides the conceptual basis for the sophisticated knowledge-based software design environments of the future. It should have the *flexibility* to support a variety of design methodologies, be *comprehensive* enough to encompass the gamut of software lifecycle activities, and be *precise* enough to provide the conceptual foundations for an open yet rigorous development environment.

We also present recent work on the structure and design of a class of algorithms called *global search*. The design tactic for global search algorithms provides a rich example of the kind of design process that the abstract model is intended to capture. We present a tactic for designing global search algorithms and illustrate it with the derivation of an algorithm for enumerating cyclic difference sets - a rare kind of set that bear some similarities to the prime numbers. The design tactic has been implemented and used to derivation dozens of global search algorithms including one for enumerating cyclic difference sets.

[illegible]
8/18/89

Contents

1 Introduction

This report summarizes recent activities on the development of a formal model of the design and evolution of software. The model is intended to be both descriptive and prescriptive. It is *descriptive* in that we are attempting to capture known design processes in the model. Examples of design processes include algorithm design tactics and optimization techniques developed at Kestrel Institute (see for example [28], system builds, version control methods, JSD (Jackson System Design) methodology [14], and Boehm's Spiral model [7]. The model is also intended to be *prescriptive* in the sense that it provides the conceptual basis for the sophisticated knowledge-based software design environments of the future. It should have the *flexibility* to support a variety of design methodologies, be *comprehensive* enough to encompass the gamut of software lifecycle activities, and be *precise* enough to provide the conceptual foundations for an open yet rigorous development environment.

There are several criteria or constraints that we feel are essential characteristics of the next generation of software development tools. We are working to ensure that these are integral to the model.

- Formal Design - to allow machine-mediation and support for the design process included the capture and reuse of design decisions
- Consistency - machine-generated changes must preserve consistency or at least some specified semantics
- Scalability - the model should apply equally to module-level design as to system-level design.
- Evolution - the model should accomodate change as an integral part of design

In Section 2 we lay out our current ideas on the nature of design and evolutionary software activities. First a conceptual model of a design state is presented and then we step back and view system evolution as a sequence of design states starting from a simple initial state. In Section 3 we present recent work on one particular design process. We have studied in some depth the structure and design of a class of algorithms called *global search* [27]. The design tactic for global search algorithms provides a rich example of the kinds of design process that the abstract model is intended to capture. In turn, the model clarifies the nature of the activities involved in algorithm design. We present a tactic for designing global search algorithms and illustrate it with the derivation of an algorithm for enumerating cyclic difference sets - a rare kind of set that bears some similarities to the prime numbers. The design tactic has been implemented and used to derive dozens of global search algorithms including the cyclic difference sets algorithm. In Section 4 we present an example of an evolution step which reflects a simple but formal elaboration of a hospital patient monitoring system.

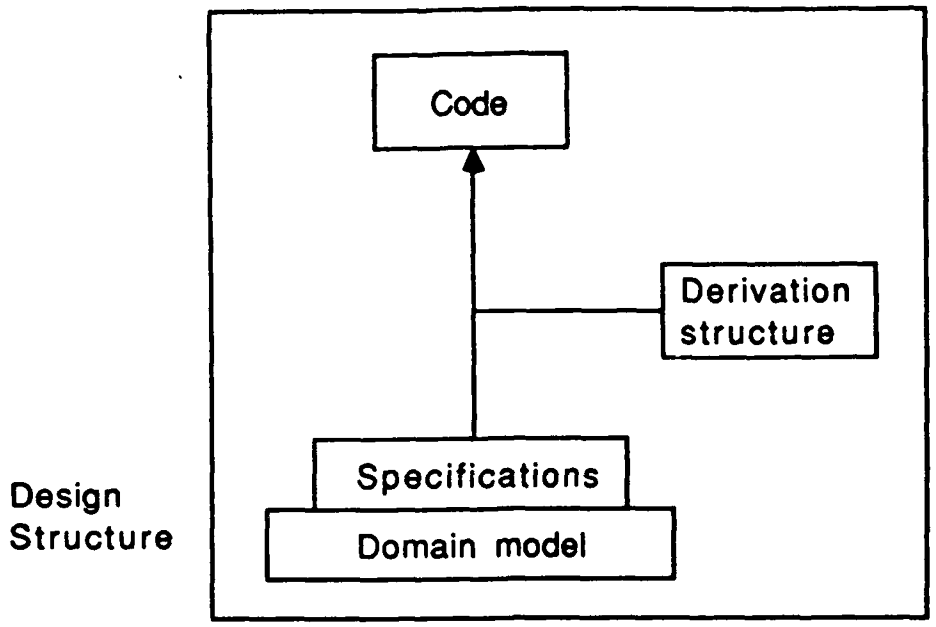


Figure 1: Design Structure

2 Toward a Formal Model of Software Design and Evolution

2.1 Design Structures

The simplified diagram in Figure 1 presents the components and relationships of our initial model of software design. We will refer to this structure as a *design structure*. The *domain model* is a formal representation of relevant aspects of the world within which the desired software is to be embedded. The *specification* component expresses constraints on the behavior of the desired software artifact. The *derivation structure* component is a record of the design decisions that connect the specifications to target code. Our model of design is a transformational one - the specifications are incrementally transformed in a stepwise refinement process into executable code that is provably consistent with the initial specifications. The final component, *code*, is a program specification expressed in some target language.

The key constraint on a design structure is mathematical *consistency* between the components. That is, the specification is stated in terms of the underlying model and is consistent with its constraints. The derivation structure provides in essence a proof that the code is consistent with the specifications - it is a proof by construction. The only assumptions used in deriving the code are those that are available in the model or in the specification itself.

For simplicity of exposition we assume a single "wide-spectrum" language for all expressions of the design structure. This in fact is the situation with our experimental system, KIDS, which is based on the REFINE language and programming environment. REFINE provides set-theoretic data types, full first-order logic constructs, object structuring mechanisms, and PASCAL-level control and data structures. Casting this assumption aside would require that our model be elaborated to include transformations between linguistic systems, as in [29].

In the following subsections we elaborate this model. In Sections 3 and 4 we present detailed examples and explain their correspondance with the model.

2.1.1 Theories

How can we formally define the notions of domain model, specification, and derivation? The remaining component, code, has by now a fairly consensual definition: programs are well-formed terms over a programming language and the programming language itself can often be expressed as a theory; that is, in terms of a collection of sorts, operators, and axioms. Could the notions of algebra and logic suffice for the other components?

We list here a few definitions of concepts from algebra [16, 10] and mathematical logic [20] that are central to our model of software design. The syntactic part of a theory T, called its *signature*, is presented by giving a collection of *sort* symbols (denoting types) and a collection of operator symbols and their arity (e.g. $f : D \rightarrow R$). The *language* of T, denoted $L(T)$, is just the well-formed terms over the signature of T. The semantic part of a theory is presented by giving a list $Ax(T)$ of *axioms.* A formula $F \in L(T)$ is a *theorem* of T if it is derivable from the axioms using the rules of predicate calculus.

A *model* of a theory is an assignment of sets to the sort symbols and functions to the operator symbols such that the arities are consistent with the sorts and the axioms are satisfied. A theory T is *consistent* if the boolean constant *false* is not a theorem of T. A theory is consistent if and only if it has a model.

Theory T' is an *extension* of theory T if $L(T) \subseteq L(T')$ and every theorem of T is also a theorem of T'. Theory T' is a *conservative extension* of theory T if $L(T) \subseteq L(T')$ and every formula of T which is a theorem of T' is also a theorem of T. Conservative extensions are often constructed by adding a new function symbol and defining axiom to a theory. Such a construction is called a *derived operator.*

A *theory morphism* between theories T and T' consists of (1) a map from the sorts of T to the sorts of T', and (2) a map from operator symbols of T to derived operations in T' such that each axiom of T is mapped to a theorem of T' when each operator symbol is replaced by its derived operator in T'. Also the operator map must be consistent with the sort map. Intuitively, a theory morphism allows us to translate the concepts and behaviors of theory T into the terms of T'. In particular this concept expresses the essential notion of implementing a abstract theory T in terms of a concrete theory T', where the terms abstract and concrete are relative.

2.1.2 Domain Model

If we want to specify and build a software system, then we need vocabulary and some expression of its semantics. In this paper we will presume a formal representation of an underlying linguistic system (such as REFINE), but will require further information about the application domain of the software system.

Domain models express the objects, operations, relationships, agents, activities, and other assumptions and properties about the application domain. It is important to explicitly represent this information because (i) it provides the vocabulary in which the requirements of the desired system are expressed, and (ii) many difficulties arise in current practice due to differing assumptions made during design.

Domain models can be classified into two kinds: static and dynamic [14]. *Static models* are used in application domains that are essentially timeless. A database provides one example of a static model. It expresses the objects and relationships of a finite world. Static models are most generally expressed as theories in classical logic. *Dynamic models* on the other hand are used when the objects and relationships can change over time. They can be expressed by theories in temporal/modal logics or process models such as state transition diagrams [13].

In this paper we limit our attention to static domain models presented as theories.

2.1.3 Specifications

Specifications describe the intended behavior of a software system. They are expressed in terms of the vocabulary provided by the domain model. Specifications consist of formal interface descriptions (services provided and required input) plus constraints on allowable behavior of the desired software and the use of the system in context. Constraints can be placed on the functionality, syntactic form, performance, reliability, fault-tolerance, etc. of the target system.

We can factor the notion of a specification into functional, structural, performance, and environmental constraints:

- Functionality deals with the logical relations between inputs and outputs (the interface with the rest of the software system's environment). Ideally the functional description is devoid of any structural constraints (implementation details). Functionality constraints describe what the system is intended to do.

- Structural constraints deal with the form of the software system, that is, how the system achieves its functional behavior. Structural constraints may describe the modules and abstract interfaces of a system, specify the use of routines from a standard library (rather than synthesizing similar code), specify the use of a certain communication protocol, etc. A LISP or ADA program can be thought of as a purely structural specification of a system.
- Performance deals with the resource utilization of a concrete program. Typical performance issues are program termination (finite consumption of resources), the amount of running time and/or memory space consumed, number of processors used, communication costs, etc. A specification might state that the target program should optimize a given cost function involving various aspects of performance.
- Environmental constraints describe the context in which the system will be used. Assumptions might describe the sizes of typical inputs, a probability measure on inputs, the relative frequency of calls on the system's utilities, number of processors available and their characteristics, etc. This information is essential in assessing whether the target code achieves its performance constraints. Environmental constraints import information from the domain model into the specification.

For the purposes of this paper we will restrict attention to specifications of the interface: functionality constraints. Specifications will be presented as conservative extensions to the domain model.

2.1.4 Derivation Structures

Derivation structures record the design decisions that connect specifications to code. Our model of design is transformational - the specifications are refined incrementally via the application of transformation rules into executable code that is provably consistent with the initial specifications. The derivation structure can be used for documenting and explaining the design and for helping to guide the design process.

The basic step in a derivation is an *implementation* step which can formalized in terms of theory extension and theory morphism. Given a specification presented as a theory $S0$, an *implementation* of theory $S0$ in theory $S1$ is a pair $\langle M, E \rangle$ where E is a conservative extension of $S1$ to $S1'$ and M is a theory morphism from theory $S0$ to theory $S1'$. See Figure 2. The intuition is that our "abstract" specification/theory $S0$ is to be implemented in terms of a "concrete" specification/theory $S1$. The theory $S1$ provides the basic vocabulary for the implementation step but we must extend it with appropriate operations and definitions so that the operations and definitions in $S0$ can be translated in a way that preserves the behavior of $S0$.

The definitions of theory morphism and conservative extension guarantee that $S1'$ is consistent with the abstract behavior/properties in $S0$.

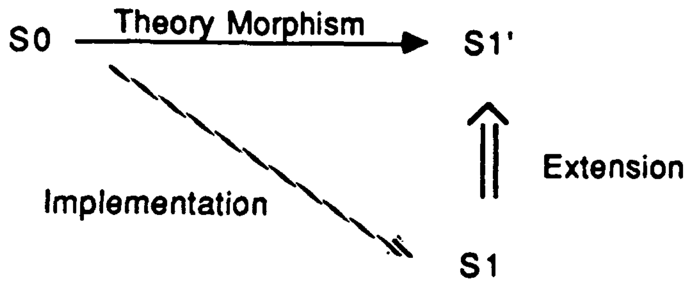


Figure 2: Implementation Step

As an example, consider the step of applying a transformation rule (conditional rewrite rule)

$$r \rightarrow s \quad if P.$$

This rule applies to specification $F[t]$ if r matches the subterm t with substitution θ and condition $P\theta$ can be verified. We conservatively extend the current specification theory by adding a new function $F'[s]\theta$ and replace every occurence of F by the new function. There is a natural morphism from the original specification into this new extended specification. As an option we could delete the unused function F and rename F' to F.

The usual model of derivation is a simple sequence of implementation steps. Implementation steps can be composed to form derivation structures. See Figure 3. However, our algorithm design tactics reveal an unexpected twist. As discussed in Section 3, we represent the abstract structure of a class of algorithms A as a 'heory, called say A-theory. The essence of designing an algorithm of a given class A is to extend the current specification theory S with the structure common to the class such that the extended theory S' is the image of a theory morphism from A-theory. Once an A-theory is created, then a concrete algorithm for the problem can be constructed. This construction is mediated by one of a collection of A-theorems that factor in commitments to target language, control strategy, and target architecture. The resulting algorithm is a derived operator that extends the problem specification. This approach to algorithm design gives the derivation structure in Figure 4. So while this design step can still be viewed formally as an implementation step, it is the target specification that is given and extended in this case rather than the source. We might distinguish these two cases of implementation calling the first a *refinement* step and the second a *design* step.

One goal of this project is to develop a theory (abstract data type) of derivations and to

$$\begin{tikzcd}
\text{Initial Specification} \arrow[r] & S0 \arrow[r] \arrow[dr, Rightarrow, dashed] & S1' \arrow[r] & S2'' \\
 & & S1 \arrow[u, Rightarrow] \arrow[r] \arrow[dr, Rightarrow, dashed] & S2' \arrow[u, Rightarrow] \\
 & & & S2 \arrow[u, Rightarrow]
\end{tikzcd}$$

Figure 3: Implementation Structure

$$\begin{tikzcd}
\text{A-theory} \arrow[r] \arrow[dr, Rightarrow, dashed] & S0' \arrow[r] & S1'' \\
 & S0 \arrow[u, Rightarrow] \arrow[r] \arrow[dr, Rightarrow, dashed] & S1' \arrow[u, Rightarrow] \\
\text{Initial Specification} \arrow[ur] & & S1 \arrow[u, Rightarrow]
\end{tikzcd}$$

Figure 4: A Design Step

validate its generality by applying it to a diversity of known implementation steps and design processes. A derivation ADT will have mechanisms for sequential and parallel composition, alternation, and iteration [30, 21]. For our purposes, mechanisms for abstraction and instantiation of derivation structure will be vital to capturing general design processes such as our algorithm design tactics. The idea is that *ground derivations* can express design histories - a trace of the decisions made (by man or machine) during a derivation, and that *parameterized derivations* express design tactics - reusable methods for deriving code from specifications. An ADT for derivations would have many of the characteristics of a metalanguage, such as ML [12], since a derivation can be viewed as a metaprogram applied to a specification to derive code.

Several other requirements on a theory of derivations that seem important from our case studies are discussed below.

Design Goals

Goals are formal descriptions of a result to be achieved. The top-level goal in a development system might specify a derivation structure that maps a given specification into a consistent source-level program specification. Other kinds of goals include obtaining certain kinds of analyses, propagating a constraint, obtaining certain kinds of inferences, gathering properties about a certain subproblem, optimizing a section of code, refining away certain kinds of constructs, etc. Goals provide the rationale for an implementation step and supporting activities. In a top-down stepwise refinement design process, the design goals also serve to focus the process and motivate the selection and application of implementation steps.

Implementation Methods

Methods are operators that perform an implementation step in order to achieve a certain kind of design goal. Methods, like all other software, have specifications of their behavior. Determining applicability of a method to a goal amounts to verifying that the method's functionality constraints will achieve or make progress towards the current goal. A method achieves a goal, either directly or by reducing it to a structure of subgoals. Direct methods may encapsulate a single transformation rule or invoke a system utility such as the compiler, analysis routine, or inference system. Other methods may spawn a complex structure of subgoals and perform arbitrary processing in combining the results of achieving the subgoals. A key question of this research is how to structure and compose methods. Analysis of our existing transformational systems suggests the need for such control structures as abstraction, alternation, sequencing, loops, and exception handling.

Goal Satisfaction and Exception-handling

An important issue is how to verify that a completed method has indeed satisfied the goal for which it was invoked. In general goal satisfaction is expressed as a theorem-proving problem, but we expect that a more efficient approach can be found. We need to be able to deal with methods that only partially satisfy the goal. In such cases, the system would need to analyze the reasons for failure and use them to suggest further action. In [24, 26] we explore the use of contingency rules for dealing with partial successes in design tactics. These rules prescribe how to fix the situation, generally by modifying the program specification to facilitate the development process. Thus the ability to detect partially solved goals seems crucial to getting a feedback relation between the design process and specification formulation process.

Content of Implementation Steps

Various kinds of implementation steps have been explored in the literature. The desired ADT should express the abstract *form* of derivations but also be expressive enough to capture the *content* of known implementation methods. We cite just a few examples of the kinds of implementation methods that our ADT should express naturally. Refinement of abstract data types has been studied extensively, e.g. [10]. Derivation methodologies such as [15, 6] rely on user-supplied implementation steps and require post-hoc verification of consistency. The abstract notion of implementation seems well-suited to support this approach. Another kind of implementation step is implied by the transformational methodology [3, 8, 2, 1]. Here there is a library of generic implementation steps (called transformation rules) and a derivation is usually treated as a sequence of rule applications. Our automated methods for designing algorithms such as divide-and-conquer [24] are yet another kind of example.

Specifying Derivation Structures

We can view the design process as a computation whose output is a derivation structure. This suggests that the design process itself can be specified and designed. That is, in general we may want to specify the design process in terms of constraints on the form, function, resource utilization, and usage of the resulting derivation structure. A typical functional constraint would be to produce a derivation structure for a program that satisfies a given problem specification. Structural constraints limit the form of the derivation, for example by using a certain derivation structure for guidance (i.e. replay). The performance constraints define the resources that can be used by the design process. Finally we might specify that the derivation structure is to be used for documentation and replay - perhaps resulting in two structures tailored to these specific uses.

2.1.5 Consistency

A key property of a design structure is its consistency. That is, treating the structure as a theory of the desired software system, it is not possible to infer a contradiction.

Inconsistency can arise in many ways. In terms of design structure we can categorize some of these. The domain model can be internally inconsistent, so tools are needed for making inferences from the model seeking contradications. Some simple necessary conditions on consistency can be tested: well-formedness of terms and formulas, and definedness of terms. The specification may be inconsistent with the domain model, but this possibility suggests that we identify and support ways to conservatively extend a domain model. A conservative extension to a consistent theory is also consistent. The way that we have defined implementation steps in terms of theory extension and interpretation between theories enforces consistency at each stage. Whatever composition mechanisms are developed for a derivation ADT must be proved to be consistency-preserving. Also any methods that are introduced to the system must be shown to be consistency-preserving. This is clearly the hardest problem, since tools are generally lacking for this task.

2.2 Evolution Structures

Software typically evolves over time – programmers must continually adapt it to meet changing needs and changing environments. Thus a successful model of design must accomodate change and evolution as an integral part of the design process.

The notion of a design structure in Figure 1 provides the framework for organizing our thinking about the evolutionary process. Initially the designer creates a simple, but consistent design. Then the designer iteratively begins transforming the design structure until a satisfactory design state is reached. These transformations are accomplished by making small but meaningful changes to either the domain model (to improve its accuracy and precision), to the specifications (to more accurately reflect the desired behavior), or to the derivation structure (to make better implementation choices). These changes are then propagated throughout the design structure according to propagation rules in order to reestablish a consistent design structure. So, for example, if we add an exceptional case to the domain model, the design system should propagate the exception into the specifications and finally through the derivation structure to be reflected in the target code. Note the shift away from the current notion of "replay" of a derivation structure on a modified specification (a kind of design-by-analogy) to the more deductive notion of propagating changes through a structure and reestablishing consistency. In a satisfactory design state the model is sufficiently accurate, the specifications have been elaborated to the point that they reflect accurately the needs of the users of the target software, and the derivation produces correct code with acceptable performance characteristics.

The key idea is that it is easier to understand, explain, build, and modify a complex object like a design structure in increments rather than all at once.

There are various ways to build a simple initial design structure. The domain model may include simplifying assumptions such as infinite memory or infinite precision arithmetic, or unbounded rationality in agents. Several data types may be confounded. The specification too may be oversimplified - perhaps dealing only with normal-case behavior and a very restricted subset of the desired functionality. The derivation structure may reflect a simple implementation strategy that yields correct executable code, but without much efficiency. Or it could implement the specification on a nonexistant very-high-level architecture. The derivation structure for a simple design state nonetheless records a derivation of code that is consistent with the model and specifications.

This model of evolution is based on the pioneering work of Goldman [11] and Feather [9] who are concerned with the evolution of specifications prior to implementation. Our approach applies the incremental elaboration idea to the entire design process and puts it on a rigorous basis.

There are several notable features of this model of design.

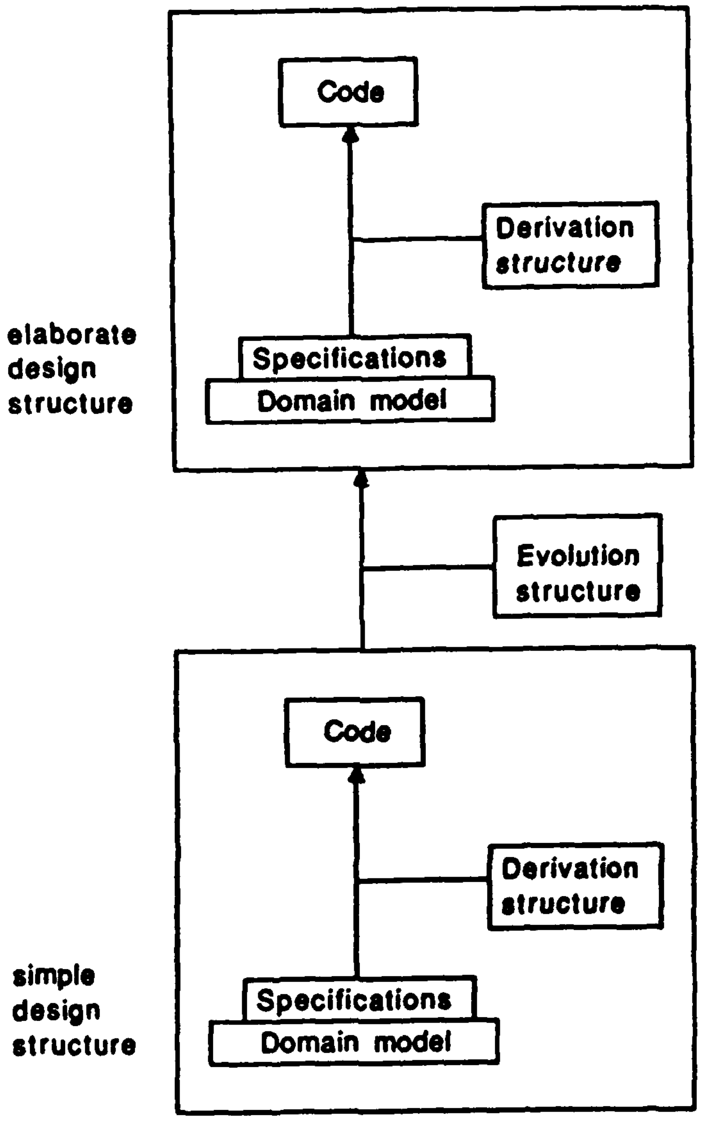


Figure 5: Evolution Structure

1. It is formal and mathematically rigorous. It is intended to only allow the production of provably correct code - at the expense of a more formal design process. Consistency is maintained at all stages of evolution.

2. Our intent is that design be entirely machine-mediated, with an increasing amount of the decision-making being automatable. Reestablishment of consistency is key to evolution and it motivates much of the machine's decision-making.

3. Design is an incremental process. Evolution and prototyping are accomodated as the basic modes of operation.

4. The model of evolution is well-structured and provides a conceptual framework for organizing software evolution activities and support systems. We can begin to catalog the kinds of changes that are typically done during evolution and the rules for propagating their effects over the components of the design structure.

5. The model is intended to apply to both system- and module-level software design.

3 Design of Global Search Algorithms

In this section we present a detailed example of a generic design method and indicate correspondance with the model of design outlined in Section 2.

Solving a problem by intelligent enumeration of the space of candidate solutions is a pervasive and well-known paradigm in the computer science and operations research communities. We explore one common enumeration method called *global search*, which applies, at least theoretically, to any partial recursive function. Global search generalizes the computational paradigms of binary search, backtracking, branch-and-bound, constraint satisfaction, heuristic search, and others.

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts candidates, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions. In our model of global search, solutions can be extracted from all nodes of the tree, not just leaves. This allows the enumeration of infinite feasible spaces where the tree is unbounded in breadth or depth.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *descriptors*. In addition to the extraction and splitting operations mentioned above, the type also includes a *satisfaction* predicate that determines when a candidate solution is in the set denoted by a descriptor. For the sake of simplifying the presentation we will use the term *space* (or *subspace*) to denote both the descriptor and the set that it denotes. It should be clear from context which meaning is intended.

The core of this section is an axiomatic representation of the abstract structure of global search algorithms. The axiomatic representation allows us to capture the essence of a class of algorithms while suppressing details of programming language, programming style, and control strategy. The main results of this section are design tactics for formally deriving global search algorithms from specifications. Being based on the axiomatic representation, the tactics are *sound* and thus they either produce correct algorithms or they fail to complete. The tactics are *abstract* and thus they are applicable across a broad variety of problems. They are also *formal* and thus are amenable to machine support. The tactics incorporate control knowledge about designing global search algorithms and thus provide *highly motivated* derivations when compared with the use of a more general but consequently weaker derivation methodology.

Our overall approach to automating algorithm design is based on formalizing classes of algorithms as abstract axiomatic theories (c.f. [17, 24]). A given problem specification is treated as a concrete theory that is a conservative extension of the underlying domain theory. Algorithm design is a process of extending the concrete problem theory with an appropriate instance of an abstract algorithmic theory. The extended theory then allows the inference of concrete programs in a variety of languages and styles (recursive vs. iterative, sequential vs. parallel, etc.). Part of the power of this approach is that much of this design process can be specified in the abstract and coded into a tactic that then applies across a broad variety of problems.

We develop a formal theory of global search algorithms called *GS-theory*. When a given problem is appropriately extended to a gs-theory then the framework of a global search algorithm is defined for it. Program parts and filters such as backtrack pruning tests, lower bound pruning tests, and dominance relations can be precisely specified and derived via inference in the extended problem theory. Each of these filters are common in known global search algorithms, but are now obtained through human invention.

We illustrate the theory by deriving a well-structured algorithm for enumerating cyclic difference sets [22, 4, 5]. Cyclic Difference Sets (CDSs) have been studied in the mathematical literature, but apparently no one has tried to exhaustively enumerate them. They are relatively rare sets and are somewhat analogous to primes in the natural numbers. The problem can be defined as follows. Given a modulus v, a set size k, and a constant ℓ, a $\langle v, k, \ell\rangle$*-cyclic difference set* C is a subset of $\{0..v-1\}$ that has size k. Furthermore, if we consider "rotating" C by adding an arbitrary constant i, where $i \ mod \ v \neq 0$, to each element yielding a new set D, then C and D have exactly ℓ elements in common. For example, the simplest CDS is the $\langle 7,3,1\rangle$-cds $\{0,1,3\}$. It has the property that for any $i \in \{1..6\}$ that

$$size(\{0,1,3\} \cap \{i + j \ mod \ 6 \mid j \in \{0,1,3\}\}) = 1,$$

for example, for $i = 4$ we have $size(\{0,1,3\} \cap \{4,5,1\}) = 1$. These sets have been used for coding satellite communications, creating masks for X-ray telescopes, and other applications. Baumert [4] collects all known CDSs whose size is less than 150. These known CDSs were found purely by mathematical construction. Below we describe the derivation of a program to enumerate CDSs. We have used this program to discover a previously unknown CDS: the $\langle 13,4,1\rangle$-CDS: {0,1,4,6}.

3.1 Domain Model

We outline here a static domain model for the CDS problem which is presented as a domain theory. First, we use the notation and semantics of the REFINE langauge which gives us standard first-order predicate calculus and set-theoretic notations, data types, and operators. In order to specify the cyclic difference set problem we must then extend this REFINE model with definitions of new terms and appropriate laws. Some of these are listed next. Unless otherwise indicated, all free variables are universally quantified.

Theory *Cyclic_Difference_Sets*

Sorts $Nat, set(Nat)$

Operations $rotate : Nat \times Nat \times set(Nat) \rightarrow set(Nat)$

$overlap\_under\_rotation : Nat \times Nat \times set(Nat) \times set(Nat) \rightarrow Nat$

$self\_overlap\_under\_rotation : Nat \times Nat \times set(Nat) \rightarrow Nat$

Axioms

$$1 \leq i \leq n-1 \ \wedge \ R \subseteq \{0..n-1\} \\ \implies rotate(i,n,R) = \{ (a+i) \bmod n \mid a \in R \}$$

$$1 \leq i \leq n-1 \ \wedge \ R \subseteq \{0..n-1\} \ \wedge \ S \subseteq \{0..n-1\} \\ \implies overlap\_under\_rotation(i,n,R,S) = size(S \cap rotate(i,n,R))$$

$$1 \leq i \leq n-1 \ \wedge \ R \subseteq \{0..n-1\} \\ \implies self\_overlap\_under\_rotation(i,n,R) \\ = overlap\_under\_rotation(i,n,R,R)$$

$$self\_overlap\_under\_rotation(i,n,\{\ \}) = 0$$

$$self\_overlap\_under\_rotation(i,n,\{0..n-1\}) = n$$

$$1 \leq i \leq n-1 \ \wedge \ a \notin R \\ \implies self\_overlap\_under\_rotation(i,n,R \ with \ a) \\ = (self\_overlap\_under\_rotation(i,n,R) \\ +overlap\_under\_rotation(i,n,\{a\},R) \\ +overlap\_under\_rotation(i,n,R,\{a\})$$

$$1 \leq i \leq n-1 \ \wedge \ R \cap S = \{\ \} \\ \implies self\_overlap\_under\_rotation(i,n,R \cup S) \\ = (self\_overlap\_under\_rotation(i,n,R) \\ +self\_overlap\_under\_rotation(i,n,S) \\ +overlap\_under\_rotation(i,n,S,R)$$

$$+\mathit{overlap\_under\_rotation}(i, n, R, S)$$

Notice that this extended theory introduces some new terminology that enables us to write a concise specification for the cyclic difference sets problem. Also a collection of distributive laws have been listed. Strictly speaking, these are redundant consequences of the earlier axioms, but we have found that distributive and montonicity laws provide most of the lemmas needed during the design and optimzation of programs. Consequently, it has become standard in our methodology to create a complete set of laws that show how a newly defined term distributes over the constructors of its input domain.

3.2 Specifications as Theories

We will treat problem specifications as first-order theories. For convenience to the reader we will also use a more conventional functional programming format for presenting specifications as a kind of "surface syntax" for the underlying specification theory.

Define *basic problem structure* $\mathcal{B}_{\mathbf{F}}$ to be a structure consisting of a function **F**, an input domain **D**, an output domain **R**, an input condition **I**, and an input/output predicate **O** plus axioms that constrain the possible denotations of function **F**. Here we restrict our attention to axiomatizing functions that find all solutions.

Theory	$\mathcal{B}_F$
Sorts	$\mathbf{D}, \mathbf{R}$
Operations	$\mathbf{I} : \mathbf{D} \rightarrow \mathit{Boolean}$
	$\mathbf{O} : \mathbf{D} \times \mathbf{R} \rightarrow \mathit{Boolean}$
	$\mathbf{F} : \mathbf{D} \rightarrow \mathbf{R}$
Axioms	$\forall x \in \mathbf{D}\ \forall z \in \mathbf{R}\ [\ \mathbf{I}(x) \implies \mathbf{F}(x) = \{z \mid \mathbf{O}(x, z)\}\ ]$

The *input condition* $\mathbf{I}(x)$ constrains the domain of values on which the desired program $\mathbf{F}(x)$ is to work. The *output condition* $\mathbf{O}(x, z)$ describes the conditions under which output domain value z is a *feasible solution* with respect to input instance x.

A *program specification* is a basic problem structure extended with an axiom of the form

$$\forall x \in \mathbf{D}\ [\ \mathbf{I}(x) \implies \mathbf{F}(x) = \mathit{Body}(x)\ ].$$

A program specification is *consistent* if the associated extended theory is consistent.

Concrete problem theories will be presented as theory morphisms from $\mathcal{B}$ into the derived operators of the particular problem.

Expressed in a more conventional format, a program specification has the general form:

```
function  F (x : D) : $set$(R)
    where  I(x)
    returns  {z | O(x, z)}
    = $Body$(x)
```

The expression *Body* (when present) is code that can be executed to compute **F**.

Example: Cyclic Difference Sets

For example, the cyclic difference sets problem can be specified in the functional format

```
function  $CDS(v : Nat, k : Nat, \ell : Nat) : set(set(Nat))$
    where  $1 \leq \ell \leq k < v$
    returns  $\{sub \mid sub \subseteq \{0..v-1\}$
                 $\wedge\ size(sub) = k$
                 $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}$.
```

and then presented as a theory morphism from basic problem structure $\mathcal{B}$ to *CDS*

$$\begin{array}{lll}
\mathbf{F} & \mapsto & CDS \\
\mathbf{D} & \mapsto & Nat \times Nat \times Nat \\
\mathbf{I} & \mapsto & \lambda\langle v, k, \ell\rangle.\ 1 \leq \ell \leq k < v \\
\mathbf{R} & \mapsto & set(Nat) \\
\mathbf{O} & \mapsto & \lambda\langle v, k, \ell\rangle, sub.\ sub \subseteq \{0..v-1\} \\
 & & \quad \wedge\ size(sub) = k \\
 & & \quad \wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}.
\end{array}$$

End of Example.

A specification is represented in terms of basic problem structure in order to later extend it with the structure of an algorithmic theory and allow inference of programs in various formats. We are concerned with those extensions that underlie the inference of global search algorithms.

3.3 Inference

Deductive inference is a pervasive activity in our approach to algorithm design. We will use a form of deduction, called ***directed inference***, that generalizes theorem-proving and formula simplification. Given some assumptions $A(\vec{x})$ and a source term (possibly a formula) $S(\vec{x})$, the goal is to find a term $T(\vec{y})$ that (1) bears a specified relationship to S, e.g. , $S(\vec{x}) \implies T(\vec{y})$ or $S(\vec{x}) \leq T(\vec{y})$ under the given assumptions; (2) satisfies some syntactic constraints, typically that its free variables $\vec{y}$ are a specified subset of $\vec{x}$; and (3) minimizes some cost function, e.g. , a heuristic measure of computational simplicity. For example, we might want to reason forward from $S(\vec{x})$ to find a consequence $T(\vec{y})$ where $\vec{y} \subseteq \vec{x}$ and

$$\forall \vec{x}\, [A(\vec{x}) \implies (S(\vec{x}) \implies T(\vec{y}))]. \tag{1}$$

In other words, we derive a necessary condition $T(\vec{x})$ on $S(\vec{x})$ under assumptions $A(\vec{x})$.

Similarly, given $A(\vec{x})$ and an integer-valued expression $e(\vec{x})$, we might want to find another expression $f(\vec{y})$ that satisfies some syntactic constraints and satisfies some comparison relation, e.g. , $\leq$, $\geq$, $=$, or $\subseteq$, with e. For example, a lower bound f for e over variables $\vec{y}$ satisfies:

$$\forall \vec{x}\, [A(\vec{x}) \implies e(\vec{x}) \geq f(\vec{y})]. \tag{2}$$

We require only that the specified direction of the inference be a preorder (i.e., reflexive and transitive).

In general a directed inference is specified as follows:

Assumptions	$A_1 \wedge A_2 \wedge \ldots \wedge A_n$
Source	$S(\vec{x})$
Inference-direction	$\longrightarrow$
Target-variables	$\vec{y}$

The inference process involves applying a sequence of transformations to the source term. The transformations are restricted to those that preserve the specified inference direction. A directed inference can be expressed

$$S = S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots S_n = T$$

with result

$$S \overset{*}{\rightarrow} T$$

where T is constrained to contain only the target variables. Some typical inference directions that will appear later include

forward inference	$\Longrightarrow$
backward inference	$\Longleftarrow$
simplification	$\Longleftrightarrow$, $=$
relaxing a set	$\subseteq$
deriving a lower bound	$\geq$

In general we will also perform forward inference from the assumptions in order to obtain a richer assumption set. That is, if we can infer B from assumptions $A_1, A_2, \ldots, A_n$ then B can be added to the assumptions. These derived assumptions often help simplify and speed up the inference process.

Deriving a necessary (resp. sufficient) condition on a formula is performed via forward (backward) inference and we will sometimes call the result a derived ***consequent*** (***antecedent***). Analogously, deriving a necessary and sufficient condition on a formula is performed via equivalence-preserving transformations and we will sometimes call the result a derived *equivalent*. In these terms theorem-proving can be viewed as the task of deriving *true* as an antecedent of a given goal formula (or of deriving *false* as a consequent of the negation of the goal). The task of formula simplification can be viewed as the task of deriving an equivalent of the given goal that minimizes some measure of formula simplicity. Much of the work in algorithm design can be treated as highly constrained directed inference. A formal system for deriving antecedents appears in [23]. We have implemented an automatic directed inference system called RAINBOW II that is invoked by CYPRESS II as a utility.

3.4 Abstract Structure of Global Search Algorithms

In this section we formalize the structure underlying global search algorithms. The formalism is of value in that (1) it allows us to clarify the nature of global search algorithms and their correctness, and (2) it provides a rigorous foundation for designing global search algorithms for particular problems. The intuitive notion of global search can be treated as the extension of basic problem structure with an abstract data type for descriptors. The operators on this type allow for creating an initial space, splitting spaces, extracting solutions, and determining element membership. Axioms constrain the possible interpretations of the extension. Formally, a *gs_extension* to basic problem structure $\mathcal{B}$ consists of the following structure:

Sorts	$\hat{\mathbf{R}}$
Operations	$\hat{\mathbf{r}}_0 : \mathbf{D} \rightarrow \hat{\mathbf{R}}$
	$\mathbf{Satisfies} : \mathbf{R} \times \hat{\mathbf{R}} \rightarrow \mathit{Boolean}$
	$\mathbf{Split} : \mathbf{D} \times \hat{\mathbf{R}} \times \hat{\mathbf{R}} \rightarrow \mathit{Boolean}$
	$\mathbf{Extract} : \mathbf{R} \times \hat{\mathbf{R}} \rightarrow \mathit{Boolean}$
Axioms	*GS*1. $\forall \mathbf{x} \in \mathbf{D}\ \forall \mathbf{z} \in \mathbf{R}\ [\, \mathbf{O}(\mathbf{x},\mathbf{z}) \implies \mathbf{Satisfies}(\mathbf{z}, \hat{\mathbf{r}}_0(\mathbf{x}))]$
	*GS*2. $\mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}) \iff \exists k \in \mathit{Nat}\ \exists \hat{\mathbf{s}} \in \hat{\mathbf{R}}\ [\, \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{s}})]$

where $\hat{\mathbf{R}}$ is the domain of meaningful descriptors, $\hat{\mathbf{r}}$, $\hat{\mathbf{s}}$, and $\hat{\mathbf{t}}$ vary over descriptors, $\hat{\mathbf{r}}_0(\mathbf{x})$ is the descriptor of the initial set of candidate solutions, $\mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}})$ means that $\mathbf{z}$ is in the set denoted by descriptor $\hat{\mathbf{r}}$ or that $\mathbf{z}$ satisfies the constraints that $\hat{\mathbf{r}}$ represents, $\mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})$ means that $\hat{\mathbf{s}}$ is a subspace of $\hat{\mathbf{r}}$ with respect to input $\mathbf{x}$, and $\mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}})$ means that $\mathbf{z}$ is directly extractable from $\hat{\mathbf{r}}$. Axiom GS1 gives the denotation of the initial descriptor — all feasible solutions are contained in the initial space. Axiom GS2 gives the denotation of an arbitrary descriptor $\hat{\mathbf{r}}$ — an output object $\mathbf{z}$ is in the set denoted by $\hat{\mathbf{r}}$ if and only if $\mathbf{z}$ can be extracted after finitely many applications of Split to $\hat{\mathbf{r}}$. Here

$$\mathbf{Split}^0(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \iff \hat{\mathbf{r}} = \hat{\mathbf{t}}$$

and for all $k \in \mathit{Nat}$

$$\mathbf{Split}^{k+1}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \iff \exists \hat{\mathbf{s}} \in \hat{\mathbf{R}}\ [\, \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}}) \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{s}},\hat{\mathbf{t}})].$$

Basic problem structure $\mathcal{B}$ with a gs-extension is called a *gs-theory* $\mathcal{G}$, and it provides the components necessary for deriving a global search algorithm.

Example: Enumerating Subsets

Consider the problem of enumerating subsets of a given finite set S. A space can be described by a pair $\langle U, V\rangle$ that denotes the set of all subsets of $U \cup V$ that extend U. The descriptor for the initial space is just $\langle\{\ \}, S\rangle$ where $\{|\ |\}$ denotes the empty map. Formally, the descriptor $\langle U, V\rangle$ denotes the set

$$\{T \mid U \subseteq T \ \wedge\ T \subseteq V \uplus U\}.$$

We present a concrete gs-theory via a theory morphism (correspondence) between the abstract components (theory signature) and object-level expressions

$$\begin{array}{rcl}
\mathbf{F} & \mapsto & gs\_subsets\_of\_a\_finite\_set \\
\mathbf{D} & \mapsto & set(\alpha) \\
\mathbf{R} & \mapsto & set(\alpha) \\
\mathbf{O} & \mapsto & \lambda S, T.\ T \subseteq S \\
\hat{\mathbf{R}} & \mapsto & \lambda S,\ \{\langle U, V\rangle \mid U \in set(\alpha)\ \wedge\ V \in set(\alpha)\ \wedge\ U \uplus V \subseteq S\} \\
\mathbf{Satisfies} & \mapsto & \lambda T, \langle U, V\rangle.\ U \subseteq T\ \wedge\ T \subseteq V \uplus U \\
\hat{\mathbf{r}}_0 & \mapsto & \lambda S.\ \langle emptyset, S\rangle \\
\mathbf{Split} & \mapsto & \lambda S, \langle U, V\rangle, \langle U', V'\rangle.\ V \neq \{\ \}\ \wedge\ a = arb(V) \\
 & & \quad \wedge\ (\langle U', V'\rangle = \langle U, V - a\rangle\ \vee\ \langle U', V'\rangle = \langle U + a, V - a\rangle) \\
\mathbf{Extract} & \mapsto & \lambda T, \langle U, V\rangle.\ empty(V)\ \wedge\ T = U
\end{array}$$

End of Example.

We will derive one recursive program specifications in gs-theory, others may be found in [27]. For an arbitrary gs-theory $\mathcal{G}_\mathbf{F}$ we can define an auxiliary function $\mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{r}})$ that yields the set of all feasible solutions $\mathbf{z}$ in space $\hat{\mathbf{r}}$; that is,

$$\mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{r}})\ =\ \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z}, \hat{\mathbf{r}})\ \wedge\ \mathbf{O}(\mathbf{x}, \mathbf{z})\}.$$

Theorem 3.1 *Let $\mathcal{G}_\mathbf{F}$ be a well-founded gs-theory. The following multilinear recursive program specification is consistent.*

```
function $\mathbf{F}(\mathbf{x} : \mathbf{D}) : set(\mathbf{R})$
    where $\mathbf{I}(\mathbf{x})$
    returns $\{\mathbf{z} \mid \mathbf{O}(\mathbf{x}, \mathbf{z})\}$
    $= \mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{r}}_0(\mathbf{x}))$

function $\mathbf{F\_}gs(\mathbf{x} : \mathbf{D}, \hat{\mathbf{r}} : \hat{\mathbf{R}}) : set(\mathbf{R})$
    where $\mathbf{I}(\mathbf{x})$
    returns $\{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z}, \hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x}, \mathbf{z})\}$
    $= \{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z}, \hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x}, \mathbf{z})\}$
       $\cup\ reduce(\cup,\ \{\ \mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{s}}) \mid \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}})\}).$
```

Proof: The main challenge is to show the consistency of $\mathbf{F\_}gs$:

$$\mathbf{F\_}gs(\mathbf{x},\hat{\mathbf{r}}) = \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}. \tag{3}$$

Assuming (3), the first specification is consistent by the following argument:

$$\begin{aligned}\mathbf{F}(\mathbf{x}) &= \mathbf{F\_}gs(\mathbf{x},\hat{\mathbf{r}}_0(\mathbf{x}))\\ &= \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}_0(\mathbf{x})) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\} \quad \text{by (3)}\\ &= \{\mathbf{z} \mid \mathbf{O}(\mathbf{x},\mathbf{z})\} \quad \text{using Axiom } GS1.\end{aligned}$$

To establish (3) we will show that

$$\lambda\mathbf{x},\hat{\mathbf{r}}.\ \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}$$

is the function computed by $\mathbf{F\_}gs$ by showing that the least fixpoint of the functional

$$\begin{aligned}\mathbf{GS}(f) = \lambda\mathbf{x},\hat{\mathbf{r}}.\ &\{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}\\ &\cup\, reduce(\cup,\ \{\ f(\mathbf{x},\hat{\mathbf{s}}) \mid \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})\}).\end{aligned}$$

In particular, if ω denotes the everywhere undefined function, then we must show that

$$\lim_{i\to\infty} \mathbf{GS}^i(\omega) = \lambda\mathbf{x},\hat{\mathbf{r}}.\ \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}.$$

First we show by induction that

$$\begin{aligned}\mathbf{GS}^i(\omega) = \lambda\mathbf{x},\hat{\mathbf{r}}.\ &\mathbf{if}\ \exists\hat{\mathbf{t}} \in \hat{\mathbf{R}}[\ \mathbf{Split}^i(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}})]\\ &\mathbf{then}\ \perp\\ &\mathbf{else}\ \{\mathbf{z} \mid k \in \{0..i\} \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}.\end{aligned}$$

The case $i = 0$ is easy since both sides evaluate to ω. Next, assume the above equality for i and consider $\mathbf{GS}^{i+1}(\omega)$.

$$\mathbf{GS}^{i+1}(\omega) = \mathbf{GS}(\mathbf{GS}^i(\omega))$$

$$\begin{aligned}= \lambda\mathbf{x},\hat{\mathbf{r}}.\ &\{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}\\ &\cup\, reduce(\cup,\ \{\ \mathbf{GS}^i(\omega)(\mathbf{x},\hat{\mathbf{s}}) \mid \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})\})\end{aligned}$$

$$\begin{aligned}= \lambda\mathbf{x},\hat{\mathbf{r}}.\ &\{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}\\ &\cup\, reduce(\cup,\ \{\ (\mathbf{if}\ \exists\hat{\mathbf{t}} : \hat{\mathbf{R}}[\ \mathbf{Split}^i(\mathbf{x},\hat{\mathbf{s}},\hat{\mathbf{t}})]\ \mathbf{then}\ \perp \mathbf{else}\ \{\mathbf{z} \mid ...\})\\ &\qquad \mid \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})\})\\ &\qquad \text{applying the induction hypothesis}\end{aligned}$$

$$\begin{aligned}= \lambda\mathbf{x},\hat{\mathbf{r}}.\ &\{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}\\ &\cup\, reduce(\cup,\ (\mathbf{if}\ \exists\hat{\mathbf{t}} : \hat{\mathbf{R}}[\ \mathbf{Split}^{i+1}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}})]\ \mathbf{then}\ \perp\end{aligned}$$

$$\text{else } \{\{\mathbf{z} \mid k \in \{0..i\} \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\} \mid \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})\}))$$

distributing the set-former over the conditional

$$= \lambda \mathbf{x},\hat{\mathbf{r}}.\ \mathbf{if}\ \exists\hat{\mathbf{t}} : \hat{\mathbf{R}}[\mathbf{Split}^{i+1}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}})]\ \mathbf{then}\ \perp\ \mathbf{else}\ \{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\} \cup reduce(\cup, \{\{\mathbf{z} \mid k \in \{0..i\} \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{s}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\} \mid \mathbf{Split}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{s}})\})$$

distributing the set union operations over the conditional

$$= \lambda \mathbf{x},\hat{\mathbf{r}}.\ \mathbf{if}\ \exists\hat{\mathbf{t}} : \hat{\mathbf{R}}[\mathbf{Split}^{i+1}(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}})]\ \mathbf{then}\ \perp\ \mathbf{else}\ \{\mathbf{z} \mid k \in \{0..i+1\} \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\})$$

simplifying

The final expression above is in the form $\mathbf{GS}^{i+1}(\omega)$. Finally, we have

$$\lim_{i\to\infty} \mathbf{GS}^i(\omega) = \lambda \mathbf{x},\hat{\mathbf{r}}.\ \mathbf{if}\ \forall i : Nat\ \exists\hat{\mathbf{t}} : \hat{\mathbf{R}}[\mathbf{Split}^i(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}})]\ \mathbf{then}\ \perp\ \mathbf{else}\ \{\mathbf{z} \mid k : Nat \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}$$

$$= \lambda \mathbf{x},\hat{\mathbf{r}}.\ \{\mathbf{z} \mid k : Nat \wedge \mathbf{Split}^k(\mathbf{x},\hat{\mathbf{r}},\hat{\mathbf{t}}) \wedge \mathbf{Extract}(\mathbf{z},\hat{\mathbf{t}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}$$

since $\mathcal{G}_{\mathbf{F}}$ is well-founded gs-theory

$$= \lambda \mathbf{x},\hat{\mathbf{r}}.\ \{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z},\hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x},\mathbf{z})\}$$

by Axiom GS2.

QED

Theorem 3.1 provides a program scheme that can be instantiated to yield a concrete consistent program for a given problem.

Example: Enumerating Subsets

Applying Theorem 3.1 to the gs-theory *gs_finite_maps* we obtain the program specification:

```
function $Subsets(S : set(\alpha)) : set(set(S))$
   $= Subsets\_gs(S, \{\ \}, U)$

function $Subsets\_gs(S, U, V) : set(set(S))$
   where $U \uplus V \subseteq S$
   returns $\{T \mid U \subseteq T \ \wedge\ T \subseteq V \uplus U\}$
```

$$\begin{aligned} = \; & \{T \mid empty(V) \;\wedge\; T = U\} \\ & \cup\, reduce(\cup,\ \{Subsets\_gs(S, U', V') \\ & \qquad\qquad \mid V \neq \{\ \} \;\wedge\; a = arb(V) \\ & \qquad\qquad\quad \wedge\, (\langle U', V'\rangle = \langle U, V - a\rangle \;\vee\; \langle U', V'\rangle = \langle U + a, V - a\rangle)\}) \end{aligned}$$

In the last line above a is bound to a single value $arb(V)$.

End of Example.

Many other algorithm specifications can also be derived in abstract gs-theory (see [27]).

3.5 Specializing a Known GS-Theory

Our tactics for designing global search algorithms rely on a directed inference system and a knowledge base of standard gs-theories for common domains. The steps of the tactic are to select and adapt a standard gs-theory, then to infer various filters, infer a concrete program, and perform optimizations on it.

We describe here how to specialize a gs-theory to a given problem specification. Let $\mathcal{G}_G$ be a known gs-theory (i.e., available in the knowledge-base) whose components are denoted $\mathbf{D}_G, \mathbf{R}_G, \mathbf{O}_G$, $\mathbf{Satisfies}_G$, etc., and let F be a given problem with components $\mathbf{D}_F, \mathbf{R}_F, \mathbf{O}_F$. The gs-theory $\mathcal{G}_G$ *generalizes* $\mathcal{B}_F$ if the feasible set of G is a superset of the feasible set of F.

$$\forall \mathbf{x} : \mathbf{D}_F\ \exists \mathbf{y} : \mathbf{D}_G\ \forall \mathbf{z} : \mathbf{R}_F\ [\mathbf{R}_F \subseteq \mathbf{R}_G \;\wedge\; (\mathbf{O}_F(\mathbf{x}, \mathbf{z}) \Rightarrow \mathbf{O}_G(\mathbf{y}, \mathbf{z}))] \tag{4}$$

Verifying (4) provides a substitution θ for the input variables of $\mathcal{G}_G$ in terms of the input variables of F. The type and number of input variables can differ between $\mathcal{G}_G$ and $\mathcal{B}_F$, as in the example below. The gs-theory $\mathcal{G}_F$ is obtained by applying substitution θ across the gs-extension of $\mathcal{G}_G$ and adding the result to $\mathcal{B}_F$. To see that the axioms GS1 and GS2 hold for $\mathcal{G}_F$ note that we have replaced the input variables of $\mathcal{G}_G$ with terms which take on a subset of their previous values.

Example: Cyclic Difference Sets.

The gs-theory *gs_subsets_of_a_finite_set* generalizes the *CDS* specification. To see this, first instantiate (4)

$$\begin{aligned} & \forall \langle v, k, \ell\rangle \in \{\langle v, k, \ell\rangle \mid 1 \leq \ell \leq k < v\} \\ & \exists S : set(Nat) \\ & \forall Sub : set(Nat) \\ & \quad [Sub \subseteq \{0..v-1\} \implies Sub \subseteq S] \end{aligned}$$

The proof is trivial and results in the substitution

$$\theta = \{S \mapsto \{0..v-1\}\}.$$

This substitution is a critical translation between the problem specified in *gs_subsets_of_a_finite_set*, which takes a single set-valued argument S, and CDS, which takes three arguments v, k, ℓ. After applying these substitutions over the gs-extension of *gs_subsets_of_a_finite_set*, we obtain the following specialized gs-theory for cyclic difference sets.

$$\begin{array}{rcl}
\mathbf{F} & \mapsto & CDS \\
\mathbf{D} & \mapsto & Nat \times Nat \times Nat \\
\mathbf{I} & \mapsto & \lambda\langle v,k,\ell\rangle.\ 1 \le \ell \le k < v \\
\mathbf{R} & \mapsto & set(Nat) \\
\mathbf{O} & \mapsto & \lambda\langle v,k,\ell\rangle, sub.\ Sub \subseteq \{0..v-1\} \\
 & & \quad \wedge\ size(Sub) = k \\
 & & \quad \wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,sub) = \ell)\} \\
\hat{\mathbf{R}} & \mapsto & \lambda\langle v,k,\ell\rangle.\{\langle U,V\rangle \mid U \in set(\alpha)\ \wedge\ V \in set(\alpha)\ \wedge\ U \uplus V \subseteq \{0..v-1\}\} \\
\textbf{Satisfies} & \mapsto & \lambda Sub, \langle U,V\rangle.\ U \subseteq Sub\ \wedge\ Sub \subseteq V \uplus U \\
\hat{\mathbf{r}}_0 & \mapsto & \lambda\langle v,k,\ell\rangle.\ \langle emptyset, \{0..v-1\}\rangle \\
\textbf{Split} & \mapsto & \lambda\langle v,k,\ell\rangle, \langle U,V\rangle, \langle U',V'\rangle.\ V \ne \{\,\}\ \wedge\ a = arb(V) \\
 & & \quad \wedge\ (\langle U',V'\rangle = \langle U, V-a\rangle\ \vee\ \langle U',V'\rangle = \langle U+a, V-a\rangle) \\
\textbf{Extract} & \mapsto & \lambda\langle v,k,\ell\rangle, ub, \langle U,V\rangle.\ empty(V)\ \wedge\ Sub = U
\end{array}$$

Theorem 3.1 allows us to assemble a recursive program specification for the feasible space of CDS:

```
function $CDS(v,k,\ell)$
   where $1 \le \ell \le k < v$
   returns $\{sub \mid Sub \subseteq \{0..v-1\}\}$
   = $CDS\_gs(v,k,\ell,\{\,\},\{1..n\})$

function $CDS\_gs(v,k,\ell,U,V)$
   where $1 \le \ell \le k < v\ \wedge\ V \uplus U \subseteq \{0..v-1\}$
   returns $\{\ sub \mid U \subseteq Sub \subseteq V \uplus U\}$
   = $\{sub \mid empty(V)\ \wedge\ Sub = U$
           $\wedge\ Sub \subseteq \{0..v-1\}$
           $\wedge\ size(Sub) = k$
           $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,sub) = \ell)\}$
      $\cup\ reduce(\cup,\ \{CDS\_gs(v,k,\ell,U',V') \mid$
                 $V \ne \{\,\}\ \wedge\ a = arb(V)$
                    $\wedge\ (\langle U',V'\rangle = \langle U,V-a\rangle\ \vee\ \langle U',V'\rangle = \langle U+a,V-a\rangle)\}).$
```

This is a correct CDS algorithm, but it enumerates all 2^n subsets and so it is not very efficient. Various ways to improve its performance are described below. A refinement decision that we will not discuss is the implementation of $a = arb(T)$; any refinement of *arb* will preserve correctness although with varying degrees of performance.

End of example CDS.

The result of the specialization process is another gs-theory $\mathcal{G}_{\mathbf{F}}$ which can be added to the knowledge base if desired, and thus reused. This approach allows the design system to be involved in the incremental acquisition of its own knowledge base. Furthermore it allows concentration of the knowledge base in specialized domains. As a simple example, a generator of mappings could be specialized to a generator of permutations, which could serve as a starting point for developing an enumerator for the k-queens or traveling salesman problems.

3.6 Program Optimization

Theorem 3.1 allows us to infer a consistent but possibly inefficient algorithm from a gs-theory. While the algorithm will be well-structured, it may allow expensive and unnecessary computation – a common situation in search algorithms. In the following sections we discuss several methods for improving the performance of global search algorithms: deriving feasibility filters, term simplification, finite differencing, data structure design and representation, and finally design of subalgorithms. Our overall design tactic suggests (but does not require) applying these transformations in the order presented, since each tends to provide opportunities for the next.

3.6.1 Filters

In this section and in Section 4.3 we define several common types of *filters* – predicates that are used to eliminate spaces from further consideration. In terms of the search tree model, filters can be used to prune off branches of the search tree that cannot yield solutions or to focus search on branches that are known to lead to solutions.

The purpose of using filters is to concentrate the search by reducing the number of spaces that are explored. To be more precise, in the Theorem 3.1 the set of spaces explored for input $\mathbf{x}$ is

$$\{\hat{\mathbf{s}} \mid k \in Nat \ \wedge \ Split^k(\mathbf{x}, \hat{\mathbf{r}}_0(\mathbf{x}), \hat{\mathbf{s}})\ \}.$$

An ***absolute filter*** is a predicate over spaces: $\psi \in map(\mathbf{D} \times \hat{\mathbf{R}}, Boolean)$. It is exploited, for example, in the multilinear recursive scheme as follows.

```
**function** $\mathbf{F}(\mathbf{x} : \mathbf{D}) : set(\mathbf{R})$
   **where** $\mathbf{I}(\mathbf{x})$
   $= \{\mathbf{z} \mid \boxed{\psi(\mathbf{x}, \hat{\mathbf{r}}_0(\mathbf{x}))} \wedge \mathbf{z} \in \mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{r}}_0(\mathbf{x}), \mathbf{z})\}$

**function** $\mathbf{F\_}gs(\mathbf{x} : \mathbf{D}, \hat{\mathbf{r}} : \hat{\mathbf{R}}) : set(\mathbf{R})$
```

$$\begin{aligned}&\textbf{where } \mathbf{I(x)} \boxed{\wedge\ \psi(\mathbf{x},\hat{\mathbf{r}})}\\ &= \{\mathbf{z} \mid \mathbf{Extract(z,\hat{r})} \wedge \mathbf{O(x,z)}\}\\ &\quad \cup\ reduce(\cup,\ \{\ \mathbf{F\_}gs(\mathbf{x},\hat{\mathbf{s}}) \mid \mathbf{Split(x,\hat{r},\hat{s})} \boxed{\wedge\ \psi(\mathbf{x},\hat{\mathbf{s}})}\})\end{aligned}$$

Since ψ is evaluated prior to each invocation of **F**_*gs*, it becomes an input invariant of **F**_*gs*. When a filter is employed as above the set of spaces explored is

$$\{\hat{\mathbf{s}} \mid k \in Nat\ \wedge\ Split^k(\mathbf{x},\hat{\mathbf{r}}_0(\mathbf{x}),\hat{\mathbf{s}})\ \wedge\ \psi(\mathbf{x},\hat{\mathbf{s}})\ \}.$$

These program specifications do not include a **returns** clause because the semantics of ψ affects the results.

The stronger the filter the fewer the spaces that are explored by the global search algorithm. That is, if

$$\psi_1(\mathbf{x},\hat{\mathbf{r}}) \implies \psi_2(\mathbf{x},\hat{\mathbf{r}})$$

(i.e., ψ_1 is stronger than ψ_2) then

$$\{\hat{\mathbf{s}} \mid k \in Nat \wedge Split^k(\mathbf{x},\hat{\mathbf{r}}_0(\mathbf{x}),\hat{\mathbf{s}}) \wedge \psi_1(\mathbf{x},\hat{\mathbf{s}})\ \} \subseteq \{\hat{\mathbf{s}} \mid k \in Nat \wedge Split^k(\mathbf{x},\hat{\mathbf{r}}_0(\mathbf{x}),\hat{\mathbf{s}}) \wedge \psi_2(\mathbf{x},\hat{\mathbf{s}})\ \}.$$

Thus, all else being equal, we are motivated to derive as strong a filter as possible. This situation is complicated by two facts however. First, the stronger filter may eliminate some spaces that contain solutions and thus not find all solutions. For example, strongest possible filter (*false*) filters out all spaces, so that the search algorithm returns no solutions. Second, the complexity of computing the filter itself has a strong effect on performance of the global search algorithm. It is possible that a weaker but cheaper filter will outperform a stronger but expensive filter.

The relationship between the semantic strength of a filter and the quality of the solution set is clarified somewhat by the following classification of filters. The question of interest when seeking feasible solutions is "Does there exist a feasible solution in a given space $\hat{\mathbf{r}}$?". Formally this is

$$\exists \mathbf{z} \in \mathbf{R}\ [\ \mathbf{Satisfies(z,\hat{r})}\ \wedge\ \mathbf{O(x,y)}\ ] \tag{5}$$

We might call this the *ideal filter* since a global search algorithm algorithm using it would explore exactly the set of spaces needed to find all solutions. However, to use this directly would usually be too expensive, so instead we use various approximations to it. These approximations can be classified as either

(i) *necessary feasibility filters*, where

$$\exists \mathbf{z} \in \mathbf{R}\ [\ \mathbf{Satisfies(z,\hat{r})}\ \wedge\ \mathbf{O(x,y)}\ ] \implies \psi(\mathbf{x},\hat{\mathbf{r}});$$

(ii) *sufficient feasibility filters*, where

$$\psi(\mathbf{x},\hat{\mathbf{r}}) \implies \exists \mathbf{z} \in \mathbf{R}\ [\ \mathbf{Satisfies(z,\hat{r})}\ \wedge\ \mathbf{O(x,y)}\ ];$$

(iii) *heuristic feasibility filters*, which bear other relationships to (5).

Necessary filters only eliminate spaces that do not contain solutions, but they may also allow spaces that do not contain solutions to be explored. So they guarantee finding all solutions but may allow unnecessary work. Sufficient filters eliminate all spaces that do not contain solutions, but may filter out some spaces that do contain solutions. Thus they do no unnecessary work, but only return a subset of solutions. Heuristic filters have the disadvantages of both — they may eliminate spaces containing solutions and allow spaces that do not contain solutions to be explored. However, a fast heuristic approximation to the ideal filter may have the best performance in practice.

If a filter ψ is derived as an approximation to the ideal filter, then it depends on both the particular problem being solved (via the feasibility conditions) and the general notions of global search (via the abstract data type of spaces). The inference of ψ is thus crucial to the construction of an effective global search algorithm. From another point of view the derivation and use of feasibility filters can be viewed as the incorporation of problem-specific information into the more general-purpose generator supplied by the gs-theory.

For several reasons we will concentrate mainly on necessary filters in this section. First, they are almost always worth deriving since they improve performance without affecting the set of solutions. Second, algorithms for optimization problems (see Section 4) are based on the ability to enumerate all feasible solutions since any one of them may be optimal. In the remainder of this subsection we elaborate on the three kinds of filters.

Necesssary feasibility filters, which will be written Φ, are defined by the condition

$$\forall x \in D\ \forall \hat{r} \in \hat{R}\ \forall z \in R\ [\ Satisfies(z,\hat{r})\ \wedge\ O(x,z) \implies \Phi(x,\hat{r})]. \qquad (6)$$

Φ can be derived via the following schematic directed inference specification:

Assumptions	$x \in D\ \wedge\ \hat{r} \in \hat{R}\ \wedge\ z \in R$
Source	$Satisfies(z,\hat{r})\ \wedge\ O(x,z)$
Inference-direction	$\implies$
Target-variables	$\{x,\hat{r}\}$

The following propositions simply state that necessary feasibility filters allow the global search schemes to compute all solutions.

Proposition 3.1 *Let $\mathcal{G}_F$ be a well-founded gs-theory. If Φ is a necessary feasibility filter then the following program specifications are consistent.*

```
function F(x : D) : set(R)
   where I(x)
   returns {z | O(x, z)}
   = {z | Φ(x, r̂₀(x)) ∧ z ∈ F-gs(x, r̂₀(x), z)}
```

```
function $\mathbf{F\_}gs(\mathbf{x} : \mathbf{D}, \hat{\mathbf{r}} : \hat{\mathbf{R}}) : set(\mathbf{R})$
   where $\mathbf{I}(\mathbf{x}) \wedge \Phi(\mathbf{x}, \hat{\mathbf{r}})$
   returns $\{\mathbf{z} \mid \mathbf{Satisfies}(\mathbf{z}, \hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x}, \mathbf{z})\}$
   $= \{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z}, \hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x}, \mathbf{z})\}$
      $\cup\, reduce(\cup, \{ \mathbf{F\_}gs(\mathbf{x}, \hat{\mathbf{s}}) \mid \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}}) \wedge \Phi(\mathbf{x}, \hat{\mathbf{s}})\})$.
```

Example: Cyclic Difference Sets.
We can obtain a feasibility filter for *CDS* by deriving a consequent over the variables $\{v, k, \ell, S, T, M\}$ according to the inference specification

Assumptions	$1 \leq \ell \leq k < v$
	$\wedge\, U \uplus V \subseteq \{0..v-1\}$
Source	$U \subseteq Sub \;\wedge\; Sub \subseteq V \uplus U$
	$\wedge\, Sub \subseteq \{0..v-1\} \;\wedge\; size(Sub) = k$
	$\wedge\, \forall(i)(\, i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}$
Inference-direction	$\implies$
Target-variables	$\{v, k, \ell, U, V\}$.

Two consequents are easily derived:

$$size(U) \leq k \;\wedge\; k \leq size(U)) + size(V).$$

From $U \subseteq Sub$ we have $U \uplus Q = Sub$ for some set Q. Then we proceed as follows.

$k = size(sub)$

$\quad = size(U \uplus Q)$

$\quad = size(U) + size(Q)$

$\quad \geq size(U)$

Thus we obtain $size(U) \leq k$ as a consequent. Analogously, we infer

$$\forall(i)(\, i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, U) \leq \ell)$$

as a consequent. Using similar reasoning from the source term $Sub \subseteq V \uplus U$ we obtain the consequents $k \leq size(U) + size(V)$ and

$$\forall(i)(\, i \subseteq \{1..v-1\} \implies \ell \leq self\_overlap\_under\_rotation(i, v, U \uplus V)).$$

```
function $CDS(v,k,\ell)$
    where $1 \leq \ell \leq k < v$
    returns $\{sub \mid Sub \subseteq \{0..v-1\}\}$
    $= \quad \{sub \mid size(\{\,\}) \leq k \leq size(\{\,\}) + size(\{0..v-1\})$
                    $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,\{\,\}) \leq \ell$
                            $\wedge\ \ell \leq self\_overlap\_under\_rotation(i,v,\{\,\} \uplus \{0..v-1\}))$
                    $\wedge\ CDS\_gs(v,k,\ell,\{\,\},\{0..v-1\})\}$

function $CDS\_gs(v,k,\ell,U,V)$
    where $1 \leq \ell \leq k < v\ \wedge\ V \uplus U \subseteq \{0..v-1\}$
        $\wedge\ size(U) \leq k \leq size(U) + size(V)$
        $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,U) \leq \ell$
                                    $\wedge\ \ell \leq self\_overlap\_under\_rotation(i,v,U \uplus V))$
    returns $\{\ sub \mid U \subseteq sub \subseteq V \uplus U\}$
    $= \{sub \mid empty(V)\ \ \wedge\ \ sub = U$
            $\wedge\ sub \subseteq \{0..v-1\}$
            $\wedge\ size(sub) = k$
            $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,sub) = \ell)\}$
        $\cup\ reduce(\cup,\ \{CDS\_gs(v,k,\ell,U',V') \mid$
                        $V \neq \{\,\}\ \wedge\ a = arb(V)$
                            $\wedge\ (\langle U',V'\rangle = \langle U, V-a\rangle\ \vee\ \langle U',V'\rangle = \langle U+a, V-a\rangle)$
                            $size(U') \leq k \leq size(U') + size(V')$
                    $\wedge\ \forall(i)(\ i \subseteq \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,U') \leq \ell$
                            $\wedge\ \ell \leq self\_overlap\_under\_rotation(i,v,U' \uplus V'))\})$.
```

Figure 6: Cyclic Difference Set Algorithm

These four consequents comprise an excellent feasibility filter. In words, they state that the partial set being incrementally constructed (U) must have at most k elements, but there must be at least k elements between U and the pool of remaining elements V. Also, the partial solution U must have a self-overlap of at most ℓ and the combined set $U \uplus V$ must have a self-overlap of at most ℓ.

Incorporating the derived filter according to Proposition 3.1 we obtain the consistent program specification in Figure 6.

End of example.

The filter Φ will often dramatically reduce the amount of work needed to enumerate the feasible space.

One feature of necessary filters is that one, *true*, is immediately available; stronger filters are obtained with more investment of computational resource at design-time.

3.6.2 Simplification and Finite Differencing

Various simplifications that exploit context can be identified in the abstract form of the program specifications in Proposition 3.1. The most obvious possibility is to simplify predicates with respect to the input conditions. For example, the predicate $\Phi(\mathbf{x}, \hat{\mathbf{r}}_0(\mathbf{x}))$ can usually be simplified with respect to the input assumption $\mathbf{I}(\mathbf{x})$. A complementary simplification is based on the fact

If $P \implies (Q \iff q)$
then $P \wedge Q$ iff $P \wedge q$.

That is, if we can simplify 'Q' to 'q' under the assumption of 'P', then we can replace the program expression '$P \wedge Q$' with the simpler, equivalent expression '$P \wedge q$'.

For example

1. The expressions $\mathbf{Extract}(\mathbf{x}, \hat{\mathbf{r}})$ and $\mathbf{O}(\mathbf{x}, \mathbf{z})$ can often be simplified with respect to assumption $\mathbf{I}(\mathbf{x}) \wedge \Phi(\mathbf{x}, \hat{\mathbf{r}})$
2. The expression $\Phi(\mathbf{x}, \hat{\mathbf{s}})$ can often be simplified with respect to assumptions
$$\mathbf{I}(\mathbf{x}) \wedge \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}}) \wedge \Phi(\mathbf{x}, \hat{\mathbf{r}}).$$

To perform one of these simplifications we set up a directed inference specification. Since $\Phi(\mathbf{x}, \hat{\mathbf{r}})$ plays such an important role in global search algorithms, we give a name to its simplified form: $\phi(\mathbf{x}, \hat{\mathbf{r}})$. It satisfies

$$\forall \mathbf{x} \in \mathbf{D}\ \forall \hat{\mathbf{r}}, \hat{\mathbf{s}} \in \hat{\mathbf{R}}\ [\, \Phi(\mathbf{x}, \hat{\mathbf{r}}) \wedge \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}}) \implies (\Phi(\mathbf{x}, \hat{\mathbf{s}}) \iff \phi(\mathbf{x}, \hat{\mathbf{s}}))]\,. \qquad (7)$$

A directed inference specification to derive ϕ is

Assumptions	$\mathbf{x} \in \mathbf{D} \wedge \hat{\mathbf{r}}, \hat{\mathbf{s}} \in \hat{\mathbf{R}}$ $\wedge\ \Phi(\mathbf{x}, \hat{\mathbf{r}}) \wedge \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}})$
Source	$\Phi(\mathbf{x}, \hat{\mathbf{s}})$
Inference-direction	$\iff$
Target-variables	$\{\mathbf{x}, \hat{\mathbf{s}}\}$

Another powerful optimization techinque is called finite differencing. There are often computationally expensive expressions remaining in the derived program that can be transformed via finite differencing techniques [18] into less expensive incremental computations. In particular, subexpressions of the filter Φ can often be incrementally maintained rather than computed each time. For examples of these techniques see [27].

3.6.3 Control strategy

The multilinear recursion of Theorem 3.1 defines a potentially infinite tree of goals (each corresponding to a space), so the control strategy must be *fair* — it must allow that each node is processed after some finite time. To ensure fairness the tactic should in general analyze the gs-theory for the depth (number of nonempty **Split** steps can be performed) and breadth (maximum number of subspaces that a space directly splits into). If it has finite depth, then a simple depth-first control suffices. If it has potentially infinite depth then in general a kind of wavefront control strategy suffices. The gs-theories listed in the Appendix all have finite depth and breadth, so program termination (and morely generally fairness) is guaranteed for the usual control strategies - depth-first, breadth-first, and heuristic search (best-first). Pearl [19] is a rich source of information on search control heuristics and their effect on performance.

3.6.4 Data structure design, subalgorithm design, compilation

High-level expressions in the algorithm can be treated in several ways. They may have sufficient algorithmic content that they can be directly compiled in to efficient code. A collection of transformation rules along the lines of those explored in the CHI project [25] could be used to perform this "compilation" process. Rules would refine the high-level control and data structures into efficiently executable lower-level constructs, perhaps motivated by performance considerations. Another possibility is to mitigate an expensive computation by introducing data structure - finite differencing is a special case. A third possibility is to subject an expensive expression to algorithm design - that is, encapsulate it and apply a design tactic to it. For example, the constraints that define extractible feasible solutions and the constraints that define the splitting step can be abstracted as specifications:

```
function $\mathbf{F\_}\mathit{extract}(\mathbf{x} : \mathbf{D}, \hat{\mathbf{r}} : \hat{\mathbf{R}}) : \mathit{set}(\mathbf{R})$
    where $\mathbf{I(x)} \wedge \Phi(\mathbf{x}, \hat{\mathbf{r}})$
    returns $\{\mathbf{z} \mid \mathbf{Extract}(\mathbf{z}, \hat{\mathbf{r}}) \wedge \mathbf{O}(\mathbf{x}, \mathbf{z})\}$

function $\mathbf{F\_}\mathit{split}(\mathbf{x} : \mathbf{D}, \hat{\mathbf{r}} : \hat{\mathbf{R}}) : \mathit{set}(\hat{\mathbf{R}})$
    where $\mathbf{I(x)} \wedge \Phi(\mathbf{x}, \hat{\mathbf{r}})$
    returns $\{\hat{\mathbf{s}} \mid \mathbf{Split}(\mathbf{x}, \hat{\mathbf{r}}, \hat{\mathbf{s}}) \wedge \phi(\mathbf{x}, \hat{\mathbf{s}})\}$.
```

3.7 Remarks on the Global Search Design Tactic

We have presented an axiomatic theory of global search algorithms and a formal method for designing concrete consistent programs from problem specifications. Our main design tactic works by specializing an existing gs-theory to the given problem. It presupposes a knowledge-base of standard gs-theories for the data types of the specification language and application domains. This knowledge base is organized as a hierarchy according to specialization relationships. In researching for this section, we derived or analyzed dozens of global search algorithms and almost all were specializations of the gs-theories that are listed in the Appendix. We have also indicated how the abstract theory could be used to directly construct a gs-theory for a given problem, although this process is less well understood and more difficult than specialization of an existing gs-theory.

The design tactics are intended to provide a highly automated tool for transforming concise problem specifications into efficient programs. The decision to design an algorithm of a certain abstract form, such as global search or divide-and-conquer, provides much structure and motivation to the design process. The creation of a gs-theory for a given problem provides the rough framework for an algorithm. A more complicated but efficient algorithm is then obtained by deriving filters, performing simplifications, finite differencing, data structure design, etc. Tactics serve to raise the level of language in which programmers can write and be assured of obtaining at least an executable prototype, if not efficiently executable code.

Global search theory and design tactics can account for a broad range of known algorithms (see Appendix). Their range of applicability is an example of an important form of reusability in software — formal knowledge of how to generate algorithms from specifications. In addition to algorithm design there are several other possible applications of global search concepts, such as data structure design, database query compilation and optimization, and the implementation of set-formers and other enumeration constructs in very-high-level languages.

4 An Example Evolutionary Design Step

This section presents a simple elaboration step in the context of evolving a specification for a hospital patient monitoring system. This work was performed by Liam Peyton, currently a Ph.D. candidate in the Computer Science Department at Stanford University.

We start with a consistent specification of a monitoring system, in which a monitor is assigned to a patient and a nurse station. The monitor tracks the value associated with a patient and if the value is unsafe, the monitor notifies the nurse station. While operational, the system is not sophisticated enough and certain elaborations are required. There may be more than one value associated with a patient which requires monitoring. Values are not intrinscially part of the patient, rather they are readings off devices which measure the values. The devices may be suspect to breakdown etc. We will illustrate our approach with one simple redesign step, namely, that we wish our model to reflect that the monitor is monitoring values that are measured by a device associated with the patient. We will make a change to our initial model and then propagate the change so as to reestablish a consistent deisgn structure. The steps described below refer to the figures in the Appendix.

There are a number of different ways of conceptualizing this type of change. One way to think of it is that concept device is introduced by splicing it in to the connections between the patient and the monitor [9] as a step towards introducing new functionality (being able to handle device breakdown). Another is that one is reorganizing the location of information within the system by moving the value from the patient to the device. The third approach is to think of the *Patient* in the old system as being a merging of two concepts (the patient, and mechanisms which measure values) a *Patient/Device* object. In the new system, the old concept will have been partitioned into two concepts *Patient* and *Device*. This will both reorganize the location of information in the system and facilitate the introduction of new functionality. We feel that this last approach is the best in that it subsumes the first two, however our redesign could have been centered around the first two appraches as well (steps 2 and 4 below). Evolution is an exploratory process in which the next overall state of the design structure emerges from smaller interacting concerns (adding a device, relocating information).

Step 1. ExtendType

This transformation simply creates *Device* as a sibling type similar to *Patient*. This change, though, by itself has no effect on the system, as none of the processing is altered. However, the semantics behind the notion of partitioning a type, can be used to drive the redesign. The information (operations) associated with *Patient* are being partitioned. This can be used to initiate a dialogue to elicit the information that would be necessary to automate the rest of the edit session. What objects that were in the *Patient/Device* class are now in the *Patient* class? In the *Device* class? Which should be split into two objects of type *Patient* and *Device*? In this case, every *Patient* will have a *Device* associated with it, so it turns out that all objects will be split into two objects of type *Patient* and *Device*. What Operations associated with *Patient/Device* will be applicable to both *Patient* and *Device*, only *Patient*, only *Device*? In this case, the operations associated with the *Patient/Device* class will be assigned to either *Patient* or *Device* not both. This is because of the fact that the *Patient/Device* class was partitioned.

Step 2. DecomposeOperation

This edit operation splices *Device* into the data flow between *Patient* and *Monitor*. In general after a transformation of this type, one has to decide whether applications of *Mpatient*, should now be *Mdevice* or *Dpatient*(*Mdevice*). Some of the objects that *Mpatient* returned might now need to be of type *Device*. However, if the information was provided initially in Step 1, then step three can proceed automatically.

Step 3. DecomposeOperationApplication or ChangeOperationApplication

This transformation changes references to *Mpatient*(x) to either *Mdevice*(x) or *Dpatient*(*Mdevice*(x)).

Step 4. MoveOperation

This step could conceivably have been initiated as part of Step 1, or it could have been the starting point in an exploration which triggered the preceding steps. A preconditon of the *MoveOperation* would be that the type *Device* has been created. As well, *MoveOperation* will create side effects that upset the internal consistency of the specification (i.e. all the operations which call *Pvalue* are calling it on objects of type *Patient*). Resolving those side effect would involve incorporating device into the data flow between *Monitor* and *Patient*.

One could think of the *MoveOperation* as a mapping that takes some of the objects from the old *Patient/Device* into objects in *Device*. Namely the ones whose *Pvalue* property is used. Knowing that the *Pvalue* property is being moved is useful in Step 1 for determining which objects should be of type *Patient* and which should be of type *Device*. It is also useful for similar reasons in Step 2.

It is generally the case in software development, that all the information needed for design is NOT available at the start of design. When it is, design should proceed automatically, when it is not, it should be possible to incorporate the new information as it is made available. Our approach is to apply transformations that are only locally consistent and complete, but may which be globally incomplete and inconsistent. However, a dependency network is maintained which keeps track of inconsistencies which need to be resolved and dependencies between edit operations. So for example, in applying Steps 1 and 2 one is not forced to immediately change the calls to *Mpatient* or determine which objects and operations should belong to *Patient* and which to *Device*. When an edit operation is applied the dependency network is updated. If any constraints are violated the dependency is added to an agenda of redesign constraints which need to be satisfied sometime before the end of the session.

Step 5. ChangeTypeOFObject

This will occur either from information gathered in step 1, or to fix a side effect of step 4. If it was fixing a side effect, it was because the dependency *Definition(Pvalue,Pvalue(p))* needed to be fixed because the constraint that p be an object of type *Device* was violated. This could either be resolved by mapping the patient p into the new object of type *Device* which has the information *Pvalue* that used to be associated with p ($Pvalue(Dpatient(p))$ or one could simply make the decision from Step 1 that p now be an object of type *Device*. Note, that making the second decision automatically makes the decision that the operation Psafe should be moved from *Patient* to *Device* as well.

Step 6. RemoveOperationApplication

As a result of Step 5, there is another side effect. Again a definition dependency has been violated $Definition(Psafe, Psafe(dpatient(Mdevice(m))))$. Note that in this case, one is undoing a decision made in Step 3. As new information becomes available it may be necessary to undo decisions. Reasoning with dependencies and constraints, using flexible transformations facilitate this process.

Controlling Design

The main difficulty in this approach is the extent to which the change propagation process can be controlled and focused. A semantic abstraction like *ExtendType* can be a useful mechanism. A tractable and decomposable characterization of the pre- and post-conditions could be useful for reasoning about transforms in a manner similar to Perrys approach to functional interface specification. Decomposable domain specific constraints and redesign goals, in a specific application domain could also be used to structure the design in a complementary fashion.

The process of design seeks to synthesize an artifact given goals and constraints which describe such things as the form and function of the artifact. A piece of software is an interconnected collection of components, with both semantic and user constraints on the interconnections. The result of design must be a consistent state, but it may be necessary to have inconsistent states along the way.

First, steps along the way may interact, thus restricting oneself to coomplete and consistent transformations may eliminate valid possible evolution steps. Conversely, disallowing inconsistent states may restrict the allowable constraints. Design is an exploratory process which results not just in a new product but a new understanding of the domain and the tradeoffs between constraints. By allowing transformations with unresolved con3traint violations, one makes explicit the relationship between the various redesign steps. One sees which steps were directly satisfying goals, and which were invoked to resolve constraint violations induced by those steps. This should make it possible to build a history hierarchy which wouldaid both in building high level redesign tactics and in facilitating incremental replay. In the case, where a specification is actually overconstrained, it should be possible to show which constraints were mutually incompatible.

Our approach is to maintain a network of dependencies associated with a specification. When a transformation (or even an edited change) occurs, the network is updated and constraints checked. Constraints violated are flagged. We maintain an agenda of goals and add goals to treat violated constraints as they arise.

5 Concluding Remarks

There has been a kind of symbiosis between the activities reported here: the development of a general model of software design and evolution on one hand, and on the other hand, the development of a design method for global search algorithms and the study of the evolution of the patient monitoring system.

The general model has helped to clarify the components of the global search design method and their interrelationships. It also poses challenging questions for the global search design method (and indeed for all design methods), such as how adequate it is with respect to evolutionary activities. That is, how do incremental changes to the underlying domain theory or to the specification propagate through the design tactic? This kind of cross-fertilization provides welcome opportunities for enriching and extending our algorithm design methods.

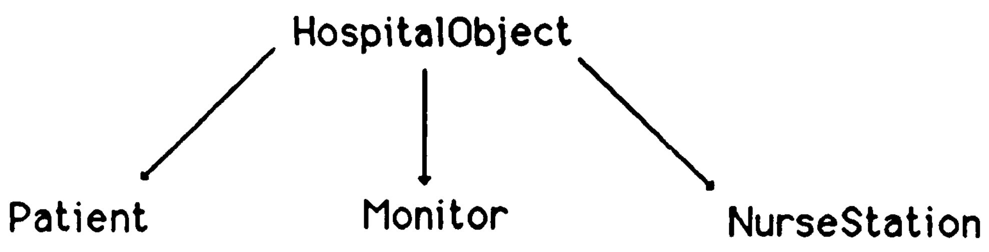
In turn, the design tactic for global search algorithms helps to validate the general model by providing a rich example of a formal software design process. The tactic (as described in Section 3) provides another level of elaboration to the general concept of derivation in the model. That is, algorithm design, simplification, finite differencing (to begin to introduce data structures), and so on.

The patient monitoring system example has suggested elaborations to our model of evolution and in turn has benefited from suggestions as to the kinds of changes that make sense from the perspective of the model.

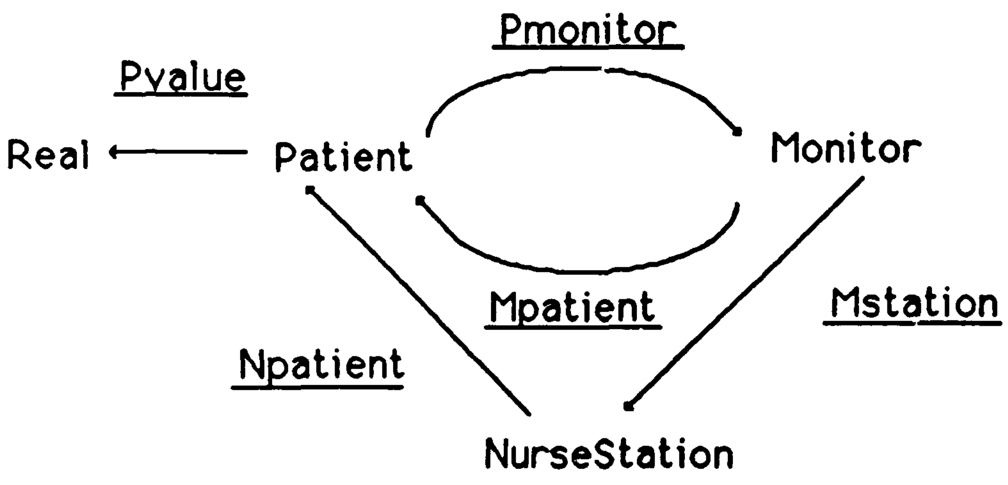
Appendix

Initial State

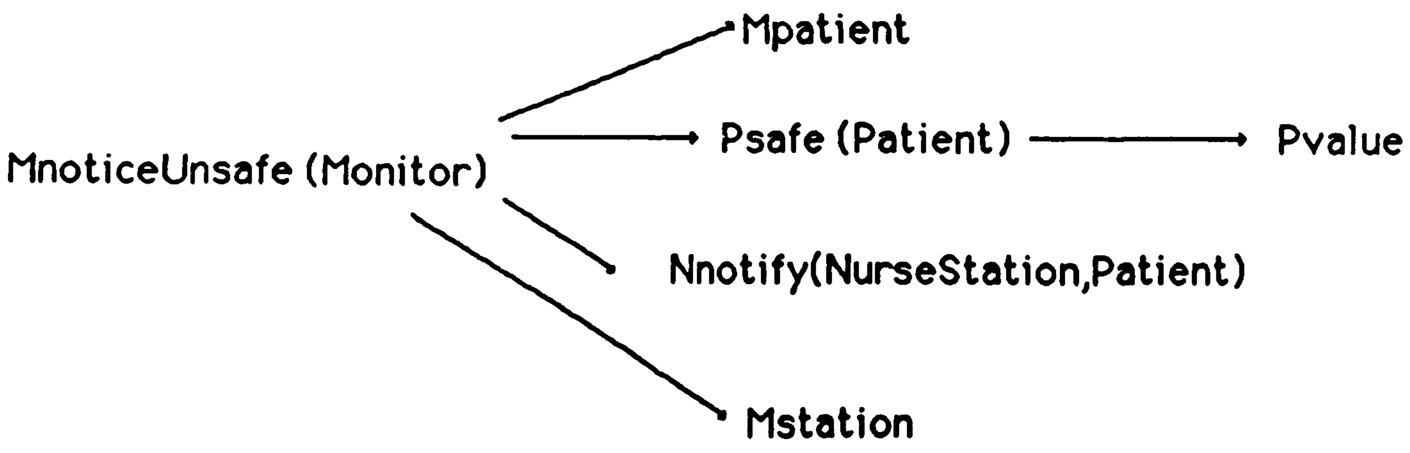
Class Hierarchy



Attribute Map

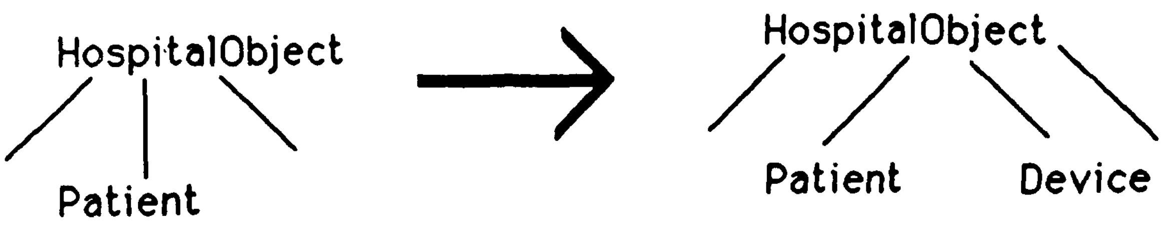


Functional Decomposition



Transformations

ExtendType



DecomposeOperation

$$\begin{tikzcd}
\text{Patient} & \text{Monitor} \arrow[l, "\text{Mpatient}"'] \arrow[r, Rightarrow] & \text{Patient} & \text{Device} \arrow[l, "\text{Dpatient}"'] & \text{Monito} \arrow[l, "\text{Mdevice}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Patient} \arrow[r, "\text{Pmonitor}"'] & \text{Monitor} \arrow[r, Rightarrow] & \text{Patient} \arrow[r, "\text{Pdevice}"'] & \text{Device} \arrow[r, "\text{Dmonitor}"'] & \text{Monito}
\end{tikzcd}$$

DecomposeOperationApplication

Nnotify(Mstation(m), Mpatient(m))

$$\longrightarrow$$

Nnotify(Mstation(m), Dpatient(Mdevice(m)))

Psafe(Mpatient(m)) $\longrightarrow$ Psafe(Dpatient(Mdevice(m)))

MoveOperation

$$\text{Patient} \xrightarrow{\text{Pvalue}} \text{Real} \quad \longrightarrow \quad \text{Device} \xrightarrow{\text{Dvalue}} \text{Real}$$

ChangeTypeOfObject

Psafe(p:Patient) $\longrightarrow$ Psafe(d:Device)

RemoveOperationApplication

Psafe(Dpatient(Mdevice(m))) $\longrightarrow$ Psafe(Mdevice(m))

Dependency Networks

$$\begin{tikzcd}
{} & \text{Device} & {} & \text{Patient} \\
\text{Pvalue} \arrow[ur, dash, "\underline{\text{DomainType}}"] & {} & \text{Psafe(p:patient)} \arrow[ur, dash, "\underline{\text{DomainType}}"'] & {} \\
{} & \text{Pvalue(p)} \arrow[ul, dash, "\underline{\text{Definition}}"] \arrow[ur, dash, "\underline{\text{Parameter}}"'] & {} & {}
\end{tikzcd}$$

Constraint Violation: Definition?(Op,Link)
=> Parameter(Link) in DomainType(Op)

$$\begin{tikzcd}
\text{Psafe} \arrow[d, "\underline{\text{DomainType}}"'] & \text{Psafe(Dpatient(Mdevice(m)))} \arrow[l, "\underline{\text{Definition}}"'] \arrow[d, shift right=2, "\underline{\text{Parameter}}"'] \arrow[d, shift left=2, "\underline{\text{Result}}"] \\
\text{Device} & \text{DpatientReturnValue} \arrow[d, "\underline{\text{ObjectType}}"] \\
{} & \text{Patient}
\end{tikzcd}$$

References

[1] BALZER, R., CHEATHAM, T. E., AND GREEN, C. Software technology in the 1990's: using a new paradigm. *IEEE Computer 16*, 11 (November 1983), 39–45.

[2] BALZER, R. M. Transformational implementation: an example. *IEEE Transactions on Software Engineering SE-7*, 1 (1981), 3–14.

[3] BAUER, ET. AL., F. L. Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming 1*, 1,2 (October 1981), 73–114.

[4] BAUMERT, L. D. *Cyclic Difference Sets*. Springer-Verlag, Berlin, 1971. Lecture Notes in Mathematics, Vol. 182.

[5] BAUMERT, L. D. Difference sets. *SIAM Journal of Applied Mathematics 17*, 4 (July 1969), 826–833.

[6] BJØRNER, D. On the use of formal methods in software development. In *9th International Conference on Software Engineering* (Monterey, CA, March 30–April 2, 1987), pp. 17–29.

[7] BOEHM, B. A spiral model of software development and enhancement. In *International Workshop on the Software Process and Software Environments* (Trabuco Canyon, California, March 27–29, 1985), pp. 22–42. (*ACM Software Engineering Notes 11(4) August 1986)*.

[8] BURSTALL, R. M., AND DARLINGTON, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (January 1977), 44–67.

[9] FEATHER, M. *Constructing Specifications by Combining Parallel Elaborations*. Tech. Rep. RS-88-216, USC/Information Sciences Institute, December 1987. To appear in *IEEE TSE*.

[10] GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, R. Yeh, Ed., Prentice-Hall, Englewood Cliffs, NJ, 1978.

[11] GOLDMAN, N. M. Three dimensions of design development. In *Proceedings of the 1983 National Conference on Artificial Intelligence* (Washington, D.C., August 22–26, 1983), AAAI, pp. 130–133.

[12] GORDON, M. J., MILNER, A. J., AND WADSWORTH, C. P. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer-Verlag, Berlin, 1979. Lecture Notes in Computer Science, Vol. 78.

[13] HAREL, D. Statecharts: a visual approach to complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.

[14] JACKSON, M. A. *System Development. International Series in Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[15] JONES, C. B. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[16] LAWVERE, F. W. An elementary theory of the category of sets. In *Proceedings, National Academy of Sciences, 50* (1963), National Academy of Sciences. Summary of PhD Thesis, Columbia University.

[17] LOWRY, M. R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13-17, 1987). Technical Report KES.U.87.10, Kestrel Institute, August 1987.

[18] PAIGE, R., AND KOENIG, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402-454.

[19] PEARL, J. *Heuristics.* Addison-Wesley, Reading, MA, 1984.

[20] SHOENFIELD, J. R. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[21] SINTZOFF, M. *Exploratory Proposals for a Calculus of Software Development.* Tech. Rep., Rapport 84/2, Universit'e Catholique de Louvain, September 1984.

[22] SKINNER, G. K. X-ray imaging with coded masks. *Scientific American 259*, 2 (August 1988), 84-89.

[23] SMITH, D. R. Derived preconditions and their use in program synthesis. In *Sixth Conference on Automated Deduction* (Berlin, 1982), D. W. Loveland, Ed., Springer-Verlag, pp. 172-193. Lecture Notes in Computer Science, Vol. 138.

[24] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43-96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[25] SMITH, D. R., KOTIK, G. B., AND WESTFOLD, S. J. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1278-1295. Technical Report KES.U.85.7, Kestrel Institute, June 1985.

[26] SMITH, D. R. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8*, 3 (June 1987), 213-229. Technical Report KES.U.85.2, Kestrel Institute, March 1985.

[27] SMITH, D. R. *Structure and Design of Global Search Algorithms.* Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[28] SMITH, D. R. KIDS - a knowledge-based software development system. In *Proceedings of the Workshop on Automating Software Design* (St. Paul, MN, August 25, 1988). Technical Report KES.U.88.7, Kestrel Institute, October 1988.

[29] TURSKI, W. M., AND MAIBAUM, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.

[30] WILE, D. S. Program developments: formal explanations of implementations. *Communications of the ACM 26*, 11 (November 1983), 902–911.